Bolt Beranek and Newman Inc.

Report No. 3983

LEVEL II

Improvements to the BCPL Programming System: Final Report

December 1978

Prepared for:
Defense Advanced Research Projects Agency

(14) BBN Report No. 3983

(12)

(6) Improvements to the BCPL Programming System

(9) Final Report: 13 apr – 30 Nov 78,
December 1978

(10) by Harry C. Forsdick,
Arthur Evans, Jr.
Robert H. Thomas

(11) Dec 78

(12) 71 p.

Sponsored by:
Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 3567

(15) Monitored by Naval Electronic Systems Command
Under Contract # N00039-78-C-0313,
VV ARPA Order-3567

Contract Period: 13 April 1978 to 30 November 1978

Principal Investigator: Robert H. Thomas

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN Report No. 3983	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Improvements to the BCPL Programming System	5. TYPE OF REPORT & PERIOD COVERED: Final Technical 4/13/78 - 11/30/78
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): H.C. Forsdick, A. Evans, Jr.	8. CONTRACT OR GRANT NUMBER(s): N00039-78-C-0313
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, Massachusetts 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: December 1978
	13. NUMBER OF PAGES: 68
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

This research was supported by the Defense Advanced Research Projects Agency under ARPA Order No. 3567.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

BCPL	inline routine call	programming language
optimization	macro	TENEX Operating System
peephole	assembler	TOPS-20 Operating System

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes BBN efforts to improve the BCPL Programming System by adding features to the language, changing the compiler and writing new routines for the run-time library. New features added to the BCPL Language include; a facility for defining routines that are expanded directly in place of calls to them; and an assembler for including PDP-10 assembly language statements in a BCPL module. Changes made to the compiler include a peephole optimizer that was added to optimize translated code and a shorter normal routine calling sequences.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Improvements to the BCPL Programming System

Contents Page

1. Summary.

The TENEX/TOPS20 BCPL Programming System is used by many ARPA contractors to implement software of interest to the Department of Defense. BCPL is favored as an implementation language for several reasons. The BCPL language allows a programmer to use convenient and powerful control structures while at the same time permitting close access to the basic efficient operations of the underlying machine. In addition, the support system surrounding the BCPL Language is complete and thus greatly simplifies the process of writing, debugging and testing BCPL application programs. BCPL is currently used as the implementation language for the System for Distributed Data (SDD-1), as well as several of the TENEX parts of the National Software Works (NSW): the Works Manager, File Package and Front End components.

The objective of this contract was to improve the BCPL Programming System by adding features to the language, changing the compiler and writing new routines for the run-time library. All of these changes are intended to accelerate the execution speed and lower the storage costs of BCPL programs. These changes and additions affect three parts of the BCPL Programming System:

1.1 Changes to the BCPL Language.

Changes to the BCPL language give the programmer better or more efficient ways of expressing algorithms. A programmer must modify existing programs to take advantage of this class of improvements. We have made the following changes to the BCPL Language:

1.1.1 Inline Routine Definitions and Calls.

A call to a routine may now be translated into a direct expansion of the body of the routine rather than a sequence of code to transfer to the body. This is beneficial for short routines where the length of the body is comparable to the length of the calling sequence.

1.1.2 PDP10 Class Assembly Language Statements.

A facility has been added to the BCPL Language for including assembly language statements in a BCPL program. This capability is intended for those special cases when the code generated by the compiler is just too inefficient for the operation being performed. These statements are necessarily machine dependent and they are clearly identified as such. In an attempt to stay as close as possible to the BCPL language, operands of assembly language statements are normal BCPL variables. Used in conjunction with inline routine definitions, sequences of assembly language statements can be packaged to take on the appearance of a normal BCPL routine call.

1.1.3 Variables Residing in Registers.

One additional change that we anticipated would be useful turned out on closer examination to be counterproductive. We originally thought that adding a facility for declaring a variable to reside in a fast register would result in an improvement in execution speed. Subsequent study showed that because of the calling sequence convention observed by BCPL, such register residing variables would have to be saved and restored on each routine call within their scope. This would eliminate any benefit gained by assigning the variable to the register. After many alternatives for providing this facility were considered, we reluctantly decided not to add this feature.

1.2 Changes to the translation of the BCPL Language.

These improvements result in more efficient translations of programs written in the existing BCPL Language (in time or space or both). Existing BCPL programs do not need to be modified to take advantage of these improvements.

1.2.1 Shortened Routine Calling Sequence.

We have implemented a shortened calling sequence for normal (i.e., not inline) routine calls. This new calling sequence maintains all of the functionality of the previous calling sequence but requires fewer generated and

executed instructions to achieve the call. Application programs run from 3% to 10% faster due to the shortened calling sequence.

1.2.2 Peephole Optimizer.

An optimizer has been added to the BCPL compiler. It is known as a "peephole" optimizer due to its method of examining a small number of machine instructions at one time for possible improvements. The peephole optimizer is table driven and as a result can be modified in the future to accommodate additional optimizations. With the current set of optimizations we have observed a 3% to 12% decrease in the size of translated BCPL modules.

1.3 Changes to the Run-Time BCPL Library.

These improvements provide better support for BCPL application programs in the form of utility routines and BCPL run-time language support. The most important additions to the library are several routines to support the new shorter routine calling sequence. Some applications will require program changes to utilize the new library routines.

In preliminary tests of small application programs written in BCPL compiled with the new compiler, we have observed a 5% to 15% overall decrease in execution time and a 5% to 15% decrease in the amount of space occupied by the machine language translation of these programs. These improvements were realized without any changes to the given application programs and as a result do not take advantage of the two changes to the BCPL Language (inline routines and machine language assembler). Further improvements will result when these two new features are integrated into application programs.

Sections 2 through 8 of this report document the changes to the BCPL language, compiler and library. In addition, Appendix A contains a detailed description of the implementation of the peephole optimizer.

2. BCPL Language Changes.

2.1 Inline Routine Definitions and Calls.

The overhead of the BCPL call, return and argument passing mechanism dominates the work done by some small routines. For such routines, it is preferable to replace the call by the actual body of the routine, adjusting references to the parameters of the routine to be references to the actual arguments of the call. This way, the only instructions generated are those corresponding to the body of the routine.

It is desirable that the semantics of such an "inline" routine and a regular routine be identical. This way, only one set of rules defining the meaning of a routine needs to be comprehended. To achieve this effect, the simple rule of having the references to parameters in the body of the routine go *directly to the arguments* supplied in the call must be changed. This is because a routine may store into its formal parameters (a common technique of providing default values for omitted arguments). In a regular routine, arguments are passed by value: the value of an argument is copied and pushed onto a stack so that the called routine has a private copy of the argument with which to work. To maintain identical semantics with regular routines, the precise rule for pushing arguments onto the stack in an inline invocation must be:

Push an argument onto the stack if:

* The argument is an expression or the reserved name "nil".

* The corresponding formal parameter is:

  * Stored into by an assignment statement (as a simple name or as the operand of the structure qualification operator "<<").

  * The operand of an "lv" operator.

  * The reserved name "nil".

Otherwise, make all references to formal parameters in the body of the inline routine or function resolve to the corresponding arguments supplied in the invocation.

The effect of this rule is to push arguments onto the stack only when necessary to preserve identical semantics with a normal routine call. It is possible to write inline routines that generate no calling sequence overhead.

The BCPL inline routine facility is designed so that changing from an "out-of-line" to an "inline" routine can be done by adding one keyword, "inline", after the "let" (or "and") of the definition:

```
   let inline Routine(Formal1, Formal2, ...) be <command>
or
   let inline Function(Formal1, Formal2, ...) := <expression>
```

Inline routines or functions are invoked the same way as regular routines or functions:

```
Routine(Arg1, Arg2, ...)
x := Function(Arg1, Arg2, ...)
```

Statements such as "return" and "resultis" jump to the statement after the inline routine or function invocation with the latter statement causing a value to be returned. The reserved name "numbargs" returns the number of arguments supplied in the inline invocation rather than the number of arguments supplied in the call to the currently executing routine.

In the following example, each time a call to the inline routines Pick, Append and Length appear, the actual body of the routine is compiled in place.

```
structure
{ String
  { N byte
    C^511 char
   }
 }

let inline Pick(CharString, Pos) :=
                CharString>>String.C^Pos

and let inline Append(Char, CharString) be
{ let N := Length(CharString) + 1
  CharString>>String.N := N
  CharString>>String.C^N := Char
 }

and let inline Length(CharString) :=
                      CharString>>String.N

let FormDirName(NameString, DirNameString) := valof
{ DirNameString>>String.N := 0
  Append($<, DirNameString)

  for i := 1 to Length(NameString) do
    Append(Pick(NameString, i), DirNameString)

  Append($>, DirNameString)

  resultis DirNameString
 }
```

Notice that inline routines may be imbedded inside other inline routines. In addition, inline routines may be defined recursively, although the decision on when to terminate the recursion must be based on an expression (probably containing the reserved name "numbargs") that may be computed as a constant at compile time. The compiler considers inline expansions that are nested to a depth greater than 200 to be erroneous infinite recursive expansions. The improvement gained by using inline routines in this example is 241 (out of 542) fewer instructions

executed per call to FormDirName when NameString is 10 characters long. Thus the inline version of FormDirName runs 45% faster than the normal routine call version.

In summary, inline routines have the same semantics as normal routines with the exception that the decision to perform a recursive call must be based on a compile-time constant expression. The primary difference between inline and regular routines is the cost of execution time and space of calls to such routines.

2.2 PDP10/SYSTEM20 Class Assembly Language Statements.

There are special cases where the advantage of using a high level language like BCPL is outweighed by the difficulty of getting the compiler to generate extremely efficient code sequences or special purpose code sequences. For these special cases, we have added to BCPL the ability to include assembly language instructions directly within a BCPL module. As a result, the need for specially coded routines in assembly language is eliminated. In addition, much of the control logic in such routines can now be expressed as standard BCPL statements.

Assembly language statements may be included in a BCPL module by use of the "assemble" statement. The syntax for this statement is:

```
assemble <MachineType>
{ OpCode(Arg1, Arg2, ...)
  ...
  OpCode(Arg1, Arg2, ...)
}
```

where <MachineType> is the name of the machine for which assembly language instructions are written. The only types of statements permitted within an assembly block are assembly language statements. The name "pdp10" describes the instruction set of the DEC PDP10/SYSTEM20 machines. Each individual machine

language instruction is expressed in the same syntax as a routine call. The names of the instructions correspond to the names of instructions found on pages 23-91 DECSYSTEM10 System Reference Manual. In addition, the names of the TENEX and TOPS20 JSYS calls (TENEX JSYS Manual and DECSYSTEM20 Monitor Calls Reference Manual) are also known by the compiler. As in the MACRO10 assembler, names of opcodes may be any combination of upper and lower case letters. The other fields of an instruction are expressed as arguments to the routine call. Following the BCPL convention, and unlike MACRO10, the default radix for numbers is decimal. JSYS calls take no arguments.

Arguments are interpreted in the following fashion:

```
OpCode()
OpCode(Address)
OpCode(Register, Address)
OpCode(Register, Address, IndexRegister)
OpCode(Register, Address, IndexRegister, IndirectBool)
```

where:

* Address is any compile-time constant or a simple variable.
* Register (IndexRegister) is any compile-time constant between 0 and #17 (octal). Note that register #16 (octal) should not normally be referenced since it is the stack pointer.
* IndirectBool is any compile-time constant which is interpreted as a truth value (true or false).

If a field is missing then the following defaults are used:

* Address is 0.

* Register and IndexRegister are 0.

* IndirectBool is false.

When a specified value is out of the range of the intended field, a warning is reported. The value put into the field is the low order bits of the specified value that will fit into the field. An example of the use of the assemble statement follows:

```
let POINT(Size, Location, RightMostBit) := valof
{ if numbargs ls 3 then RightMostBit := -1

  assemble pdp10
  { SETZM(2)
    MOVE(1, Size)
    LSHC(1,-6)
    MOVEI(1, 35)
    SUB(1, RightMostBit)
    LSHC(1,-6)
    HRR(2, Location)
    Move(1, 2)
   }
 }
```

Notice that the names referenced in the address portion of the assembly language instructions are normal BCPL variables. The alternation between upper and lower case in the last instruction is used to illustrate the fact that opcode recognition is case independent.

While instructions are expressed syntactically as routine calls, the semantics of such routine calls are somewhat different. As in a routine call, when an argument to an assembly language instruction is a constant, manifest or truth value, then

its value is put into the appropriate field. When an argument in the Address position is a simple variable (local, static, global or external) then the address of the argument (lv) is put into the appropriate field. For static, global or external variables, this is a relocatable address. For local variables, the offset of the variable on the stack is placed in the address portion and register #16 (octal) is put into the index register portion of the instruction. Attempts to reference a local variable in the Address field of an instruction that contains an explicitly specified index register are considered by the compiler as errors. These rules of field interpretation preserve the conventions present in the MACRO-10 assembler, but depart from normal BCPL argument evaluation rules.

The "address versus value" rule makes it difficult to manipulate literal constants because the Address argument is always viewed as the position of data rather than the value of data. To facilitate manipulation of literal data by assembly language instructions, a new operator "literal" has been added to the language. This operator may be used only within an assemble block as a unary operator on the Address field. Thus to generate a reference to the value 3,,4 (i.e., #3000004) rather than the address 3,,4 (which is the constant #3000004 or, truncated to 18 bits, 4), the following statement could be written:

MOVE(1, literal 3,,4)

Space would be allocated to hold the value 3,,4 so that when the instruction is executed that value is loaded into register 1.

Finally, labels may be used in the same way they are used in normal BCPL code, with identical scoping rules. It is possible to jump out of an assemble block into BCPL code, but because of the normal scoping rules for labels, it is impossible to jump into an assemble block from BCPL code. To achieve the same effect, an assemble block can be split into two assemble blocks the label affixed to the second block.

Transferring to a label from within BCPL code is done by the "goto" command. Transferring from within an assemble block requires knowledge of how BCPL labels are implemented. With one exception, a label on a statement results in the declaration of a manifest value which is the relocatable address of the start of the statement. The exception is for a label which is also declared to be a global or external. For such labels, the address of the associated statement is stored in a cell in static storage and the value of the label is the address of this cell. The impact of this on assembly code is that transfers to regular labels from within an assemble block can be done by direct transfers to the label name while transfers to global or external labels must be done by indirect transfers. A safe procedure is to exit the assemble block and use the BCPL "goto" statement.

The BCPL assembly language facility allows a programmer access to features of the underlying machine and operating system that the compiler does not know about or cannot utilize efficiently. Assembly language statements have been added in the spirit of the rest of the BCPL language; normal BCPL variables are used as the values of the fields of instructions. Because of the obvious machine dependency, assembly language statements should be used only as a last resort, when all other possibilities have been exhausted. When possible, assembly language sequences should be packaged as abstract operations by use of inline routines. This way, the complex semantics associated with raw machine language instructions can be hidden from casual BCPL programmers.

2.3 Variables Residing in Registers.

A change that we anticipated would be useful turned out on closer examination to be counterproductive. We originally thought that adding a facility for declaring a variable to reside in a fast register would result in an improvement in execution speed. Several alternatives for expressing and implementing this mechanism were considered. The most promising one is outlined as follows:

A variable may be declared to reside in a fast register. There are three ways to declare a register variable:

1. let register <VariableName> := <Expression>

   for example,
   ```
   let register x := 45
   ```

2. register
   { <VariableName> }

   for example,
   ```
   register
   { x }
   ```

3. for register <VariableName> := <Expression> to
   <Expression> by <ConstantExpression>

   for example,
   ```
   for register x := 1 to N do ...<use of x>...
   ```

The first declaration is intended for situations where the register variable is to be both declared and initialized and where its scope is limited to a small area of a BCPL module. The second declaration is intended for cases where the register variable is to be used throughout the entire module (or a set of modules) and is initialized once and referenced from many places. The third statement is both a declaration and a command and has the same meaning as a normal "for" loop command, except that the loop variable is kept in a register. The scope of a "register { <VariableName> }" statement is the same as the scope of a "static { <VariableName> := <Value> }"

statement. The scope of a "let register <VariableName> := <Value>" statement is the same as a normal "let <VariableName> := <Value>" statement. The scope of the loop variable in a "for register" statement is the statement of the for loop.

The compiler will assign registers to be used to hold variables and will refrain from using these registers for code generation purposes during the scope of the declaration.

There are two immediately problems with this proposal and all of the other proposals we considered concerning the allocation of registers:

1. If as stated, the compiler makes register assignments, then separately compiled modules run the chance of conflicting over the dedication of a register to hold a variable. Various schemes for helping the compiler manage the register assignment task were considered. The only solution that does not put too great a burden on programmers seems to be to have the compiler save registers across routine or function calls.

2. There are two ways to save registers across function calls: either on the caller's side or on the callee's side. Saving registers on the caller's side would probably negate the effect of storing a variable in a register in the first place. Saving on the callee's side only if the callee used variables in registers is an attractive solution at first. However, on further consideration this solution lengthens the routine calling sequence and thus, for those routines which use

register variables, negates the effect of the shortened calling sequence.

The language C(1) does offer such a capability for storing variables in registers. Unlike BCPL, C already saves registers across routine calls and thus saving registers used to store variables offers no additional load on the calling sequence.

After many alternatives for providing this facility were considered we reluctantly decided not to add this feature.

(1) Ritchie, D. M., et. al., "The C Programing Language," The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, pp. 1991-2019.

3. BCPL Compiler Changes.

The following descriptions of changes to the compiler are intended for readers interested in how the new language features were implemented. In addition, there are sections on the implementation of the new calling sequence intended for programmers of applications that contain features which imbed knowledge of the calling sequence. (This is a practice which in general we discourage.) Finally, the implementation of the peephole optimizer is described for compiler implementers and other interested parties.

Most BCPL programmers will not need to use any of the information contained in this section.

3.1 Inline Routine Definitions and Calls.

Inline routines are implemented by replacing the call to the inline routine by a copy of the body of the routine and then compiling the body. There are three main tasks to be performed: storing the body of the routine, copying the body of the routine to replace the call and examining the argument/parameter pairs of the call and the definition to see how to pass arguments.

All routines (inline or regular) are defined in a context of static, global, manifest, structure and external names. This environment must be carried along with the copies of the routine

so that when a copy is compiled in place of a call, a free reference to a name will resolve to the proper instance of the name. A free reference to a name in a routine should go to the instance defined at the point of routine definition, not at the point of the call. For example, in the following program fragment, the free reference to the name "String" in the inline routine Append refers to the structure name "String" rather than the local name "String" defined just before the call to "Append".

```
structure
{ String
  { N byte
    C^511
  }
 }

let inline Append(Char, CharString) be
{ let N := Length(CharString) + 1
  CharString>>String.N := N
  CharString>>String.C^N := Char
 }

<...>

let String := vec 511

<...>

Append($<, String)

<...>
```

To do this, inline routines must be scanned at the point of their definition and all references to free variables must be bound to the definitions in effect at inline routine definition time. These bindings are stored in the parse tree of the body of the routine. For inline routines, the value associated with the name of the routine is a pointer to the root of this parse tree.

The replacement of the call by the body of the routine is done in terms of the parse tree. The parse tree of the inline routine serves as a template for the expansion. The part of the tree represented by the call is replaced by a copy of the template.

In regular routine calls, all arguments are pushed onto a stack and the called routine references its corresponding formal parameters from these locations. For inline routines, argument passing is done according to the rule stated in section 2.1, which is restated below:

> Push an argument onto the stack if:
>
> * The argument is an expression or the reserved name "nil".
> * The corresponding formal parameter is:
>   * Stored into by an assignment statement (as a simple name or as the operand of the structure qualification operator "<<").
>   * The operand of an "lv" operator.
>   * The reserved name "nil".
>
> Otherwise, make all references to formal parameters in the body of the inline routine or function resolve to the corresponding arguments supplied in the invocation.

Thus, the decision whether to push an argument onto the stack is a function of both the nature of the argument and the way the formal parameter is used within the body of the routine.

Before the body of the inline routine is compiled in place of the call, each of the parameters is declared as a local variable. The nature of the location which stores the value of that variable depends on whether the argument is pushed onto the stack. If it is, then the location associated with the local parameter variable is the stack location. If not, then the location is an indirect pointer back to the actual argument. When the inline routine body is actually compiled, references to the formal parameters will either resolve to the locations on the stack that resulted from argument pushes or the locations associated with the arguments themselves. Once the routine template has been copied and the arguments have been processed, the copy is then compiled as if it had been the call.

3.2 PDP10/SYSTEM20 Class Assembly Language Statements.

The BCPL assembler facility is implemented by using reserved routine names which correspond to the opcodes of the instructions. Since assembly language statements are only legal within an assemble block, these names are only reserved in the context of such a block. In fact, the reserved names of the opcodes may be used freely outside of an assemble block. By adopting the same syntax for an assemble statement as for a routine call, changes to the BCPL parser were minimized.

Within an assemble block, each time a routine call is encountered, the arguments are examined and built up into the fields of an instruction. If the unary operator "literal" appears before an Address field, then the value contained in that field is stored in static literal storage. The address containing the value is put into the Address field. Erroneous values for fields are detected and reported. The translation of each routine call is a single machine instruction. Within an assemble block, statement types other than routine calls are noted as errors in the program.

The assembler is written so that it would be easy to implement an assembler for any language similar to the PDP10/SYSTEM20 class assembly language.

3.3 Shortened Routine Calling Sequence.

In the past, the BCPL calling routine calling sequence performed ten different tasks split between the caller's side and the callee's side. For a function, this results in 14+N instructions being executed, where N is the number of arguments to the function. For a routine call, this number is 12+N. Specifically, the steps for a function call are:

1. Load arguments to the function onto the stack, including the number of arguments to the function and whether or not the function was called from the left side of an assignment statement. (at least 2 + number of arguments, caller)
2. Transfer to the called function. (1 instruction, caller)
3. Adjust the stack frame pointer to point to a new frame above the caller's frame on the stack. (1 instruction, callee)
4. Save the return address to the caller. (1 instruction, callee)
5. Perform the stack cover computation. (3 instructions, callee)
6. Perform the stack overflow check. (3 instructions, 2 always executed, callee)
7. After the body has executed, load the return value into the return register. (1 instruction, callee)
8. Return to the called routine. (1 instruction, callee)
9. Adjust the stack frame pointer to point to the caller's frame. (1 instruction, caller)
10. Store the value of the function. (1 instruction, caller)

Specifically, the instructions generated for the following function call and definition are:

```
Z := Function(X, Y)

        MOVE    Ø,N(16)         ; Current size of frame (1).
        PUSH    Ø,[Ø,,2]        ; [leftside,,numbargs] (1).
        PUSH    Ø,X(16)         ; Arg 1 (1).
        PUSH    Ø,Y(16)         ; Arg 2 (1).
        JSP     1,@Function     ; Transfer to function (2).
        SUBI    16,N-1(16)      ; Get old frame (9).
        MOVEM   1,Z(16)         ; Store value of function (1Ø).

let Function(A, B) := valof { ... }

Function:.+1                    ; Here + 1.
        ADDI    16,@Ø(1)        ; Make new frame (3).
        MOVEM   1,Ø(16)         ; Save return address (4).
        MOVEM   17,1(16)        ; Save old cover (5).
        MOVEI   17,S(16)        ; Load new cover S (5).
        CAMG    17,16           ; Wrap around stack check (6).
        CAMLE   17,GL8Ø57       ; Beyond end stack check (6).
         JSP    1,@GL8Ø58       ; Overflow handler (6).

        <Body of function>

        MOVE    1,Resultis      ; Load resultis value (7).
        MOVE    17,1(16)        ; Restore old stack cover (5).
        JRST    @Ø(16)          ; Return (8).
```

Five of the 12+N instructions executed are due to the stack cover and stack overflow computations.

In the new compiler, the stack cover and stack overflow computations that were done on every routine or function call have been replaced by mechanisms that require no instructions to be executed in the calling sequence. The corresponding instructions in the calling sequence above (the six instructions of steps 5 and 6) have been eliminated. The overall improvement in the execution speed of an application program due to shortening the calling sequence depends on the number of routine

calls that occur, but typically 5 to 8% improvements in execution speed have been observed.

3.3.1 Stack Cover Computation.

In the previous compiler, the stack cover was maintained to be the exact length of the stack frame of the routine or function currently executing. This value is used to determine where to put a new stack frame when a PSI (an interrupt) occurs. The new frame is used by a routine which has been set up to field the interrupt. In the new compiler, the stack cover is defined to be the maximum stack excursion of any BCPL routine in the set of BCPL routines in an application program (i.e., the set of BCPL routines in a single .EXE or .SAV file). In the new scheme, forming a new frame by adding the stack cover to the current stack frame pointer will always yield a safe frame, if not the lowest unused frame on the stack. In addition, we have provided routines to adjust the stack cover should a particular application need to do so.

The new compiler keeps track of the maximum stack frame length (MSFL) for all of the routines in a single module. The stack cover is then, just the maximum of the MSFLs for all of the BCPL modules loaded together into an application program. To compute this value, it is necessary to examine the MSFLs either at the time the BCPL modules are linked together or sometime

during the running of the application program. The program LINK on DECSYSTEM2Ø does not support such a capability at link time. It does have a feature which allows BCPL modules to be chained together at link time, so that the chain can be traced at runtime. At the end of each BCPL module, the following sequence (expressed in assembly language) is emitted:

```
Link1:  BLOCK   1
        <Maximum Stack Frame Length>
        .LINK   1,Link1
```

The last line directs LINK to add the label "Link1" to the developing chain number "1". When LINK encounters the item type generated by this statement, it stores the current value of the head of chain number "1" into the location labeled "Link1" and sets the current value of the head to be the address "Link1". In one routine in the BCPL support library, there is the following sequence:

```
GL8140Ø:  BLOCK   1
          .LINK   -1,GL8140Ø
```

When LINK encounters the item type generated by the last statement, it remembers after all of the modules have been loaded to store the last value of the chain head in the location labeled "GL8140". Thus, at runtime, the location labeled "GL8140" contains a pointer to the start of a chain of pointers. The word after each of these pointers is the MSFL for each BCPL module in the set of modules linked together. The last pointer in the chain has the value Ø.

The function GetMaxStackFrameSize() will traverse the chain and return the maximum value encountered. The routine SetStackCover(MaxStackFrameSize) will set the stack cover to MaxStackFrameSize. These two routines are called in sequence during the initialization of every BCPL application program.

3.3.2 Stack Overflow Check.

In the new compiler, instead of checking on every routine call to see if the stack would overflow if the called routine were to execute, a trap is set to detect attempts to reference beyond the end of the stack. This is done by dedicating a page of memory (or set of pages) above the top of the stack to be read only. These pages may be used by the application program to store read only data. Attempts to extend the stack into this area will result in a read only protection violation which will halt the program. It is possible to write a routine which would enable a PSI to field the read only violation and determine if the trap was due to a stack overflow or to some other reason.

The routine SetStackEnd(StackEnd, NumberReadOnlyPages) will make the page(s) above the value of StackEnd to be read only. Currently, the default value of the stack is #776777 (octal). SetStackEnd is called as part of the BCPL initialization.

Most BCPL programs will not require any changes due to these two changes in the implementation of the routine calling sequence.

3.4 Peephole Optimization

This section describes the modifications made to the BCPL compiler to examine the compiled code and make improvements to it: The Peephole Optimizer. The method used is modeled after that used in the Bliss-11 compiler.(1)

3.4.1 Overview.

As the code is compiled, it is emitted as a threaded list of nodes, the format being described in detail below. Three such chains are produced, one for each segment of the object program: the code segment, which contains all code that is to be executed; the impures segment, which contains static data items declared by the BCPL static declaration; and the literals segment, which contains only character string literals that appear in the source code.

The term "peephole optimization" as used in this report refers to eight specific optimizations that are performed by passing over the compiled code. This is repeated until no more improvements result. Rarely are more than three passes made. The eight optimizations are as follows:

(1) Wulf, W., et. al., "The Design of an Optimizing Compiler," Elsevier, New York, 1975.

- Initial. An initial pass is made over the code to perform three tasks. First, certain instructions are changed to equivalent forms. This is not really an optimization because it has no effect on the execution of the program. However, the PDP-10 hardware has an extremely rich collection of opcodes providing, in many cases, multiple ways to accomplish the same task. Replacing equivalent instructions by a single form makes later parts of the optimizer simpler. Second, RETURN instructions are collected. BCPL compiles the command JRST @0(16) to return from a function or a procedure. The first instance of this instruction is noted and all subsequent instances are replaced by a JRST to that one. The Cross Jump optimization (see below) will further modify this replacement. Third, any instruction which is a no-operation is deleted.

- Alteration. Replace certain specific code sequences by better ones. (It is this specific optimization that is often called "peephole optimization" in the literature.)

- Cross Jump. If the instruction before a JRST is identical to the instruction that precedes the label to which the JRST leads, replace the former by a JRST to the latter. Similarly optimize two identical sequences ending in a JRST to the same label. In both of these

situations, the two threads are merged as far back as possible.

- Unreachable. After a JRST which is not preceded by a skip, delete all code up to the next following label, since such code can never be reached.

- Jump Chain. If a conditional jump or a JRST leads to a JRST, then the address of the first is changed to that of the second.

- Unused Label. Any label not referred to anywhere is deleted.

- Label Merge. Adjacent labels with no intervening code are merged into a single label.

- Literals. Delete un-referenced strings.

All of these optimizations except the last one operate on only the code segment; the last one affects only the literal segment.

The remainder of this section briefly outlines the implementation strategy for the peephole optimizer. Readers with no interest in implementation should skip to the next section.

On completion of the TRN phase, CGAssemble is called to complete the generation of the .REL file. It calls

PeepHoleOptimize, which first does the peephole optimization itself and then calls PeepFinal to pass over each of the threads to assign final PC values and perform other bookkeeping tasks. (If a listing file has been requested, it is emitted after optimization and before Final.) Finally, CGAssemble does the work of emitting the .REL file.

The work of optimization is performed by routines in the new module 1ØPPPG. The routine PeepHoleWork, called from PeepHoleOptimize, is the driver. It first calls PeepInit to make an initial pass over the code segment. It then makes repeated passes over the code segment, where each pass consists of calling each of three specific optimization routines. Passes continue until no further improvement is made in a pass. Finally, a single pass is made over the literals segment to delete strings not used in the program.

There are two ways in which optimization is suppressed in certain cases. If an instruction is marked SIC (by supplying an optional extra argument to the routine CGCode that compiles instructions), then that instruction is not changed in optimization. Further, marking an instruction with the CJChain bit means it is not altered in Cross Jump (see below) unless the preceding instruction is also altered.

As mentioned earlier, a compiled instruction may be marked in either of two ways to suppress optimization. Marking it SIC suppresses its possible alteration or deletion in PeepInit, keeps it from being part of an alteration sequence in Alteration, and keeps it from being changed in Cross Jump. SIC is used for two reasons: The debugger BDDT expects the first instruction in each module to be a JFCL (which is a no-operation) whose address is the beginning of the module's static area. This instruction is marked SIC since otherwise it would be deleted as a no-operation in PeepInit. Secondly, assembler instructions included by the user in his program, using the new "assemble" feature described earlier in this report, are marked SIC. Here the decision is that we do not want to attempt to second-guess the programmer.

The CJChain attribute is used in only one case. The BCPL subroutine calling sequence requires that the instruction immediately after the JSP to the subroutine be a SUBI to adjust the stack pointer. Therefore the SUBI must not be altered in Cross Jump unless the JSP is also, and so the SUBI is marked CJChained.

The peephole optimizer first calls PeepInit to perform the initial pass. Then it makes repeated passes over the compiled code performing all but the last of these optimizations in turn on each pass, continuing until no further improvement is made in a pass. DoAlter performs Alteration, DoCrossJump performs

Cross Jump, and DoJump performs Unreachable, Jump Chain, Unused Label, and Label Merge. Finally, DoLiterals makes a single pass over the literals segment deleting unused strings.

The optimizations interact with one another in various ways. For example, note that Unused Label and Label Merge make no change in the compiled code. However, performing these in one pass often makes Alteration possible in the next pass in a place where it was not earlier permitted. Note also that Crossjump and Jump Chain create unreachable instructions, which are later found in Unreachable. Label Merge permits Cross Jump to be more effective since there are more chains leading to the same point. Similarly, collecting all RETURNs to a single place (in PeepInit) increases Cross Jump's ability to find common sequences. After this is done, Jump Chain changes the instructions back to the usual RETURN.

Appendix A contains more details about the implementation of the peephole optimizer.

4. BCPL Runtime Library Changes.

Four new routines have been added to the BCPL Library as a result of the improvements to the compiler. Three of the routines are part of the BCPL initialization action while the forth implements the subcommand scanner documented in section 6.

4.1 Routines to Support BCPL Initialization.

The following two routines are used to support the new method of computing the stack cover:

Integer := GetMaxStackFrameSize()

> Returns the maximum length of a stack frame for all routines in a group of modules linked together as an application program.

SetStackCover(Size)

> Sets the value used as the stack cover to be Size. The stack cover is a value that will be used to increment the stack frame pointer yielding a new (unused) address on the stack. This address will serve as the base of a frame for a routine that is to process a PSI (interrupt).

The following routine is used to support the stack overflow checking mechanism.

SetStackEnd(StackEnd[, NumberReadOnlyPages])

Define the end of the stack to be the last address on the page containing the address StackEnd. The read only protection mechanism is used to implement this check. A page is created above the page holding StackEnd and the access is set to be read only. If a routine call results in extending the stack into this page (or set of pages), a read only violation will occur. The second argument, if present, will be used to create a set of read only pages NumberReadOnlyPages long. The default for this value is 1.

4.2 Routine to Support Command Scanning.

The following routine is used as a command scanner in the compiler. It is a general purpose routine that can be used in many applications that require user specified commands.

```
(CmdCode,,TermChar) :=
      GetWord(StreamIn, StreamOut, Commands, Separators[,
              PrefixString[, PromptString[, SuffixString[,
              CmdString[, SkipInitialSeparators[,
              HorizontalHelp]]]]]])
```

A routine to get a command from a keyboard device. GetWord is intended to implement many of the common desired functions of a command scanner including such features as command recognition and completion and limited help in determining the menu of permissible words to type. Characters typed will not be examined by GetWord until a separator is typed. When a separator is typed, the characters typed so far are examined to see if they uniquely identify one of the command strings in Commands. If they do, then GetWord returns the value (CommandCode,,TermChar), where CommandCode is the value that corresponds to the item typed and TermChar is the separator typed to end the command.

The character "?" may be typed at any point and the set of possible commands that begin with the characters already typed will be printed on the output stream. The partially typed word will be redisplayed along with any prefixes or prompts. Command recognition is case independent. The

characters "^A", Backspace and, on TOPS2Ø "DEL" will all delete one character from what is already typed. The characters "^W" will delete and "^R" will retype what has been typed so far. The characters "DEL" on TENEX and "^U" on TOPS2Ø will type " XXX " on the output stream and cause GetWord to return (-1,,Ø).

The arguments to GetWord are:

StreamIn -- The stream from which to take characters.

StreamOut -- The stream on which to echo characters.

Commands -- A vector of CommandCodes and CommandStrings, one pair per word of the vector in the format (CommandCode,,CommandString). CommandCode is normally an integer code less than #2ØØ (octal) which corresponds to CommandString, a BCPL character string which is the command.

If CommandCode is greater than or equal to #2ØØ (octal) it is assumed to be the address of a routine that should be called to determine if a unique command that it recognizes has been typed. For this case, the CommandString should describe the type of item the Routine will recognize. This way, "?" will show a generic description as one of the possible commands. The call to the routine is:

```
Routine(OpCode, CharVec[, CmdString, StreamOut])
```

OpCode is either gwComplete (manifest equal to 1) or gwCouldBePrefix (manifest equal to 2). If OpCode is gwComplete then the optional arguments will be present and Routine should type on StreamOut the completion of the unique command typed so far The characters typed so far are contained in CharVec, with CharVec|Ø indicating the number of characters in CharVec, one character per entry. In addition, the entire command should be copied into CmdString. If OpCode is gwCouldBePrefix, then Routine should return a Boolean value indicating whether or not what has been typed could be the start of a command that Routine recognizes.

Commands|Ø, if non-zero, is the count of entries in the vector. Commands|i, 1<=i<=(Commands|Ø) is the ith (CommandCode,,CommandString) pair. If Commands|Ø is zero, then the end of the Commands vector is marked by a zero entry.

Separators -- A vector of characters that are to act as command separators and command completion characters. Typing one of these characters will cause GetWord to attempt to match what has been typed so far with the list of Commands. In addition to this set, the character Escape will always cause GetWord to look at what has been typed.

Separators|Ø is the count of entries in the vector. If Separators|Ø is zero, then a zero entry terminates the list of characters.

PrefixString -- A string which will be typed first if the typed line needs to be retyped (either by ^R or by ?). Default is the empty string.

PromptString -- A string which will be typed on StreamOut to prompt the user before any characters are accepted from StreamIn. If the typed line needs to be retyped, PrefixString and PromptString will be typed, in that order, before the typed commands. Default is the empty string.

SuffixString -- A string which will be typed after command completion occurs. Default is the empty string.

CmdString -- The entire completed command will be copied into CmdString after a command has successfully been recognized. Default is not to copy the completed string.

SkipInitialSeparators -- A boolean value which, if true, indicates that initial instances of the separators, before any non-separators characters have been typed, should be discarded. Default is false.

HorizontalHelp -- A boolean value which, if true, indicates that when "?" is typed to request a menu of possible commands, these commands should be typed in a horizontal format rather than one command per line. Default is false.

5. Performance Tests.

We are currently in the process of conducting performance tests of the new BCPL compiler on both the BCPL compiler itself and the NSW Works Manager and File Package components.

Initial tests with small programs that make no use of the new language features indicate that execution speed improvements will be in the range of 5% to 15% due to both the peephole optimization and the shorter calling sequence. Since the density of calls in different application programs varies widely, improvements due to the shorter calling sequence will also vary widely.

When performance tests on the NSW components are completed, we will issue a technical report documenting our results.

6. How to Use the New Compiler.

The new compiler is called "BCPL 4.1.X", where X is a small integer indicating versions of the "4.1" compiler. Subsequent modifications to correct bugs will result in successive versions.

Any questions, comments or bug reports should be sent to:

BCPL@BBN

or

Harry Forsdick
Bolt Beranek and Newman, Inc.
50 Moulton St.
Cambridge, Ma. 02138

telephone:
(617) 491-1850

6.1 New Language Features.

New features to the BCPL language described in section 2 may be used by including them in programs.

6.2 New Compiler Features.

A new user interface had been added to the BCPL compiler. In the old user interface, options to the compiler were specified by single character flags of the form "/x" where "x" was associated with some option. In the beginning these flags corresponded to the first word of a sentence describing the option, as in "/o" standing for "old calling sequence". As the

number of flags increased, the correspondence was more and more strained. The new user interface permits options to be specified as longer character strings which are more descriptive of the options being specified. Since there are still some programs that require single letter flags to drive the compiler, use of this new user interface is completely optional. All options to the compiler can still be expressed as single letter flags.

The compiler is invoked by issuing the command "bcpl":

```
@bcpl
BCPL 4.1.2 12/04/78 09:11:50
<Short message>
Type "?" for help.
=>
```

The response identifies the compiler and prompts the user for a command line terminated by a carriage return. The syntax for a command line is:

```
    <SourceFile>[<Switches>][,]
or
    <RelFile>=<SourceFile>[<Switches>][,]
or
    <ListFile>,<RelFile>=<SourceFile>[<Switches>][,]
```

Items is square brackets are optional.

Like the BCPL compiler on TENEX, and unlike previous BCPL compilers on TOPS20, it is possible to type the entire command line on the same line as the "bcpl" command. Thus,

```
@bcpl test
```

will compile the program test. If a listing file is specified in the command line, the output will go to a file whose extension is .MAC rather than .LST as was true with the old compiler.

If the command line to BCPL is terminated by a comma, then the subcommand scanner is entered to examine and specify options to the compiler. This command scanner is implemented by the GetWord routine in the BCPL library. In subcommand mode, the compiler will prompt the user for commands with the symbol "==>". A menu of valid commands may seen by typing "?". Character delete and line delete work as expected. "^W" deletes and "^R" retypes what has been typed. The commands are long enough so that they are self describing. Command recognition is case independent. Only that portion of a command needed to disambiguate it from all other commands needs to be typed. Any character from the set {Space, Escape, Carriage Return, Line Feed} will terminate the command. The command "go" will cause the compiler to compile the indicated file with specified options.

As an example, here is a typescript of the compilation of a file Test.BCP:

```
@BCPL
BCPL 4.1.2 12/04/78 09:11:50
Test compiler - assembler, inlines, new calling sequence,
  new user interface, peephole optimizer.
=> TEST,
==> ?
 Show chosen compiler options
 Messages to .LOG file
 Save parse output
 Symbol table produced
 Long symbol table
 Peephole optimization
 Cross jump peephole optimization
 Old calling sequence
 Stack overflow check
 Stack cover maintained
 Upper case input file
 Code generation
 Debugging enabled
 List parse tree
 List lexemes as scanned
 Compiler debugging
 Set default switches to current setting
 No
 Go
==> MEssages to .LOG file
==> GO
[BCPL: TEST.BCP]
@
```

The parts of the commands actually typed are in uppercase.

6.3 The Peephole Optimizer.

The peephole optimizer is controlled by the "/g" switch to the compiler and by the "Peephole optimization" command to the subcommand scanner. The default is for the peephole optimizer to run. The cross jump optimizations that may be performed by the peephole optimizer are controlled by the "/h" switch and by the

"Cross Jump" command to the subcommand scanner. The default is for cross jump optimizations not to be performed because they interfere with the correct operation of BDDT. Programs that are compiled for production versions should use the cross jump optimizations.

Because of the new method of storing the instructions translated by the compiler, the maximum size of a BCPL module that can be compiled has been reduced. Nevertheless, modules containing 1000 lines of BCPL code have been compiled successfully with the new compiler.

6.4 *The New Routine Calling Sequence.*

The calling sequence has changed from the previous compiler. With one exception, the new calling sequence is upward compatible with the old calling sequence. The one exception is that old BCPL modules will not contain the LINK item type that is used in constructing the chain of BCPL .REL files. Thus, if an application does not use PSIs, then old modules can be loaded in with modules compiled with the new BCPL compiler, as long as the first module loaded is one compiled by the new compiler. Normally this will mean that the module containing the routine Start be the first routine specified to LINK.

The new compiler will generate the old calling sequence if the "/o" switch or the "Old calling sequence" subcommand is specified.

Cautious users of BCPL will recompile all modules with the new compiler to avoid any confusion.

6.5 The New library.

The entire BCPL library has been recompiled with the new compiler or made compatible with the new calling sequence for those routines written in languages other than BCPL. The new library is called "BCPLB3.REL". Each module compiled by the new compiler will generate a LINK item "Request" for "SYS:BCPLB3.REL". Modules compiled with the "/o" switch or "Old calling sequence" subcommand will contain requests for "SYS:BCPLB2.REL"

6.6 BDDT -- The Debugger.

Except for cross jump optimizations (discussed above), BDDT works correctly on code produced by the new compiler. The "print" statement will print the text of an inline expansion rather than the inline routine call.

6.7 The Concordance Generator.

The utility program, CONCORDANCE, has been updated to accept and examine the new BCPL language features.

7. Additional BCPL Utility Programs.

7.1 DMPREL

A new utility program has been written to produce a readable symbolic listing of a standard REL file. The program DMPREL is cognizant of the standard DEC format for REL files under TENEX and TOPS-20. It reads such a file and produces its content into an output text file. The program knows about LINK types Ø to 37 and lists each one in an appropriate format. The program proved useful to us in debugging changes made to the compiler and will be made available to all BCPL installations. Figure 1 shows a fragment of the output of DMPREL for a REL file.

```
<FORSDICK>CLUTIL.REL.11,30-Nov-78 08:41:46

  0  Entry(4)      4  GL3203  GL3202  GL3201  GL3200

  6  Name(6)       2  CLUTIL  ASCII3 000000

 12  Code(1)      22  000000 000022*
                      255000 000147*     0    JFCL    147*
                      271721 000000      1    ADDI    16,@0(1)
                      202056 000000      2    MOVEM   1,0(16)
                      201016 000006      3    MOVEI   6(16)
                      261000 000160*     4    PUSH    160*
                      261016 000003      5    PUSH    3(16)
                      265060 000000      6    JSP     1,@0
                      275700 000005      7    SUBI    16,5
                      202056 000005     10    MOVEM   1,5(16)
                      326040 000014*    11    JUMPN   1,14*
                      201100 000001     12    MOVEI   2,1
                      202116 000005     13    MOVEM   2,5(16)
                      550116 000002     14    HRRZ    2,2(16)
                      302100 000002     15    CAIE    2,2
                      254000 000026*    16    JRST    26*
                      201016 000007     17    MOVEI   7(16)
                      261000 000161*    20    PUSH    161*

 36  Code(1)      22  000000 000021*
                      261016 000005     21    PUSH    5(16)
                      261016 000004     22    PUSH    4(16)
                      265060 000000     23    JSP     1,@0
                      275700 000006     24    SUBI    16,6
                      254000 000033*    25    JRST    33*
                      201016 000007     26    MOVEI   7(16)
                      261000 000160*    27    PUSH    160*
                      261016 000005     30    PUSH    5(16)
                      265060 000023*    31    JSP     1,@23*
                      275700 000006     32    SUBI    16,6
                      202056 000006     33    MOVEM   1,6(16)
                      201016 000010     34    MOVEI   10(16)
                      261000 000162*    35    PUSH    162*
                      261016 000003     36    PUSH    3(16)
                      261016 000006     37    PUSH    6(16)
                      200116 000006     40    MOVE    2,6(16)
                      270116 000005     41    ADD     2,5(16)

 62  Code(1)      22  000000 000042*
                      275100 000001     42    SUBI    2,1
                      261000 000002     43    PUSH    2
                      265060 000000     44    JSP     1,@0
                      275700 000007     45    SUBI    16,7
                      200056 000006     46    MOVE    1,6(16)
                      254036 000000     47    JRST    @0(16)
                      271721 000000     50    ADDI    16,@0(1)
                      202056 000000     51    MOVEM   1,0(16)
                      201016 000003     52    MOVEI   3(16)
                      261000 000163*    53    PUSH    163*
                      261000 000163*    54    PUSH    163*
                      261000 000163*    55    PUSH    163*
```

Figure 1
Sample Output of DMPREL

7.2 GLINDX

The program "GLINDX" produces several listings of names declared to be global. This is useful in managing global numbers in large application programs. The listings are sorted in several different ways, including sorted on global numbers, on the names of globals, and on class of use of the name.

GLINDX prompts for an input file and an output file. The input file is a text file with one entry per global name. An entry for a global name must obey the following syntax:

<GlobalNumber>: <GlobalName> <DefModule> <DecModule> <Class> <CR>
<PrototypeCall>; <Description>

where

<GlobalNumber> is a global number of the form GL4526.

<GlobalName> is a legal BCPL name corresponding to the global number.

<DefModule> is the name of the source file that defines the name.

<DecModule> is the name of the head file that declares the correspondence between the name and the global number.

<Class> is the general grouping under which the name belongs. Examples of classes might be "Strings" and "Files".

<CR> is Carriage Return.

<PrototypeCall> is a general representation of how to call the routine. For example,
"NumberSkips := JSYS(jsNumber[, ACsIn[, ACsOut]])" is a prototype call of the JSYS function, with "[]" used to indicate optional arguments.

<Description> is a short description of the purpose of the function.

The output file from GLINDX can be printed on the line printer by the COPY command. Figure 2 shows a fragment off the output of GLINDX for sorting the list by name.

```
Global Names, Declaring Module, Description and Prototype Call   Page 42

Global Name                DecMod       Description
  Prototype Call


PBOUT                      Head         Write byte to OUTPUT (not same as
                                        PBOUT JSYS!)
  PBOUT(Byte)

PMAP                       ForkHead     Map a page into address space
  PMAP(Source, Destination[, Access])

POINT                      Head         Return a byte pointer
  BytePtr := POINT(Size, Location[, RightMostBit])

PrintF                     Head         Print on line printer
  PrintF(FormatString, Arg1, Arg2, ..., ArgN)

PrintStream                UtilHead     Stream to TTY:
  BOUT(PrintStream, Char)

PSIChDis                   PSIHead      Disable given PSI channels
  PSIChDis(Chan1[, Chan2[, ... ChanN]...])

PSIChEnb                   PSIHead      Enable given PSI channels
  PSIChEnb(Chan1[, Chan2[, ... ChanN]...])

PSIChInit                  PSIHead      Cause an interrupt on given channels
  PSIChInit(Chan1[, Chan2[, ... ChanN]...])

PSIClear                   PSIHead      Clear all outstanding interrupts
  PSIClear()

PSIOff                     PSIHead      Turn PSI system off
  PSIOff()

PSIOn                      PSIHead      Turn PSI system on
  PSIOn()

PSISetCh                   PSIHead      Assign PSI channel to routine
  PSISetCh(Level, Channel, Routine)

PSIStackOverflow           NotDeclared  Routine to transfer to when stack
                                        overflows on PSI
  JSP 1,PSIStackOverflow
```

Figure 2
Sample Output of GLINDX

8. Additional Documentation.

All of the on-line documentation about changes to the BCPL language and changes to the BCPL library have been collected and merged into two separate message files known as:

<BCPL>BCPL-Changes.TXT
and
<BCPL>BCPLIB-Changes.TXT

These will be maintained in the future to contain any incremental additions to the BCPL Programming System. All other files of on-line documentation about BCPL are now obsolete.

This leaves the documentation of the BCPL Programming System far from complete. Future documentation on the BCPL Programming System should include the following manuals:

* BCPL Language Reference Manual.
  This manual would document all features of the language in a precise manner. The first seven chapters of the current BCPL Manual could serve as a model for the information contained in this document.

* BCPL Primer and User's Guide.
  New users of BCPL have difficulty comprehending the model of the programming environment provided by BCPL. The first part of this manual should serve as an introduction to writing

programs in BCPL. The second part should act as a general guide for writing programs in BCPL and should suggest some programming standards as well as illustrate some complete examples of an application program built out of BCPL modules.

* BCPL Runtime Library User's Guide.

  Every routine in the BCPL Library should be documented to tell its purpose, its arguments and its return values. In addition, an example of a call to each routine should be given. Finally, a cross reference list of routines showing which head files declare them and which routines they in turn reference should be produced.

* BDDT User's Guide.

  The section of the current BCPL manual on BDDT, the debugger, should be rewritten to reflect the current state of commands to BDDT. In addition, a new section on debugging techniques should be included. Finally, a comprehensive example debugging session should be presented and annotated.

* BCPL Utility Programs User's Guide.

  All of the programs that are in current use to aid the process of building and maintaining BCPL programs should be documented in a similar format.

Improving the documentation of BCPL is the most important advance that is currently needed in the BCPL Programming System.

Appendix A: The Implementation of the Peephole Optimizer.

This appendix gives additional details about the implementation of the peephole optimizer. It is intended for readers interested in specific details about the implementation of the BCPL peephole optimizer as well as those interested in general techniques for implementing a peephole optimizer.

A.1 Data Formats

There are three data formats to be described: the threaded compiled code, the table PeepTable, and the table of optimization sequences used by Alteration.

A.1.1 Threaded Code

Code as it is compiled is emitted in a threaded list form. Each node in the thread holds all data about a single instruction or label or data item. (If a listing file has been requested, additional nodes provide relevant data.) Each node holds pointers to both the previous node and the following one, so that it is easy to delete a node or insert one during optimization. A label appearing in the output is threaded into place just before the instruction it labels. The label node has fields pointing to the first and last node in a Reference Chain, a threaded list with a RefNode for each instruction that references the label. The address part of each such instruction points to a RefNode in

that chain, which in turn holds a pointer back to the code node that references it and a pointer to the label node. The effect is that all references to a label can be readily found, as is clearly needed in Crossjump. A label not referred to has an empty Reference Chain.

The effect of all of these pointers is suggested by Figure 3. This shows a section of the code segment including a label and three JRSTs whose address is that label.

A.1.2 PeepTable

This table contains a one-word entry for each PDP-10 opcode, from #100 to #677. An additional entry #77 is used in DoAlter as if LABEL were an opcode. The format of PeepTable is described by the structure Peeps in 10PPHD. The left half of each word is used only by PeepInit and the relevant fields are described as part of the description of that routine. The remaining fields are as follows:

- MaySkip bitb
  The opcode may (when it is executed) cause the next instruction to be skipped.
- MayJump bitb
  The opcode may cause a jump.
- Class bit 6
  This field defines the class the opcode falls into, as used inn DoAlter.

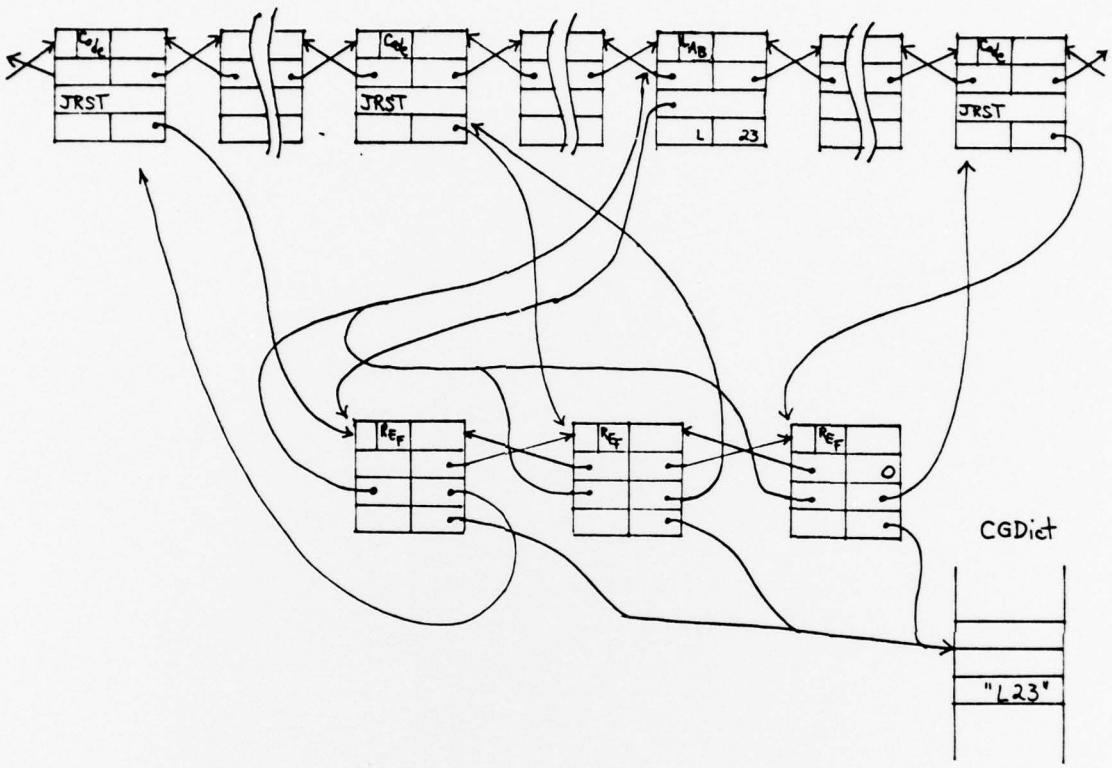

Figure 3
Threaded Code Format

. AlterTable bit 6
This specifies the beginning of a table of alteration sequences whose last member has the opcode of this entry. The entry is the offset in the table AlterTable (appearing at the end of 10PPST) whose entries contain the address of the alteration tables themselves. This added indirection requires only six bits in PeepTable rather than an 18-bit address.

A.1.3 The Alteration Tables

The Alteration optimization consists of replacing certain sequences of instructions by better ones. The routine is table-driven in the sense that the sequences to be replaced and the changes to be made in them are stored in tables. It is these tables that are described in this section.

The tables themselves are found in the module 10PPST (PDP-10 peephole statics), in the latter part of that file. Consider the optimization

MOVEM R,X + MOVE R,X ==> MOVEM R,X

which is #2 in the listing. It indicates that if a MOVEM is followed by a MOVE with the same register and address fields, the latter may be deleted. This sequence is represented in memory by five words. The first is a header word, with the Head bit set. The number 2 in the left half indicates that it is optimization #2, and the 2 in the right half indicates that the match is on two instructions. (Inclusion of the optimization number was extremely useful in debugging.) The next two words describe the

instructions to be matched, and next two describe the changes to be made in them if the match succeeds. In the present case, all that is required of the first word is that its opcode be MOVEM. The second must have opcode MOVE, and its register and address fields must match those of the first word. The next two words indicate that the first instruction is to be unaltered and the second is to be deleted.

An experienced programmer should have little difficulty deducing the significance of the remaining possible fields. The manifests used to create these tables are declared in 1ØPPST immediately following PeepTable. These manifests should be compared with the structure declaration of Alters in 1ØPPHD and with the code in the routines DoAlter, CheckMatch, and AlterNode.

A.2 Details of the Algorithms

This section presents a brief overview of some of the algorithms used. There are two purposes to this discussion: First, a reader knowledgeable about compilers and somewhat familiar with BCPL will be able to understand what we have done so as to modify other compilers similarly. Second, an experienced BCPL programmer familiar with the internals of the compiler will find this information useful in maintaining or modifying the code. For the first purpose, only this overview should be necessary. Programmers interested in the second

purpose will readily obtain additional details through study of the listings. They are extensively commented.

A.2.1 Creating the Listing File

In the compiler as it was before the present modification, each routine that compiled code tested a switch and, if required, outputed data to the listing file. With the addition of optimization, it is necessary to delay production of the listing file until after all optimization has been performed. Maintaining the integrity of the listing file (in the sense that it correspond exactly with the .REL file) was felt to be necessary for two reasons: programmers requesting the listing have a right to count on its matching the .REL file, and the listing file was an important tool in debugging the optimization routines.

Since the listing file is created after optimization, it is necessary that all data needed to create it be in the three threads. (There a few exceptions.) Therefore additional node types provide the necessary data. Nodes of these types are created only if a listing file is requested by the user in invoking the compiler. The listing is printed in PeepHoleOptimize after performing all optimizing and before calling PeepFinal.

A.2.2 The Optimization Routines

Five routines perform the optimizations themselves. They are all called from PeepHoleWork and are all found in 1ØPPPG. They are described in the following five subsections.

A.2.2.1 PeepInit -- Initial Pass

This routine performs the initial over the code, making the three changes already described: collecting all RETURN instructions, altering certain instructions to equivalent forms, and deleting instructions which are effectively no-operations. The last two optimizations are under the control of bits in PeepTable. The structure declaration Peeps in file 1ØPPHD (PDP-1Ø peephole head) describes this table. The fields relevant to PeepInit are now described, where for each switch (item declared bitb) the comment describes the action taken in PeepInit if the switch is true.

- Del bitb
  Delete the instruction.
- DelACZero bitb
  Delete the instruction providing that the AC field is zero.
- Alt bitb
  Alter this instruction by replacing its opcode by the one in the NewOp field (below).
- AltACZero bitb
  Alter the instruction only if the AC field is zero, the alteration being as above.

- NewACZero bitb
  Set the AC field to be zero.

- NewAdrZero bitb
  Set the address field to be zero.

- NewOp bit 9
  If Alt or AltACZero is set, this is the new opcode for the instruction.

The above fields are examined only in PeepInit.

A.2.2.2 Alteration

The most complicated of the optimization routines is DoAlter, which scans the code chain looking for code sequences which it can replace by better ones. As it scans, it maintains a window of the last (up to) seven instructions scanned. After each new instruction is entered into the window, the window is matched against a table of optimization sequences, with a replacement being made if a match is found. To keep the matching process from being too time consuming, it is performed in two steps. First, the opcodes of the instructions in the window are matched against the opcodes of the optimization sequences. Only if this cheap test is passed is the more time consuming test performed for exact match. The test for exact match is performed for one instruction by the routine CheckMatch. Once a match has been found, the routine AlterNode makes the actual changes.

The optimization sequences are arranged by the last opcode in the sequence. Thus, for example, all sequences ending in the opcode MOVE are in a single table. Each entry in PeepTable includes a field which points to the table of optimization sequences ending on that opcode, the field being zero if there are no such sequences. The field is named Peeps.AlterTable and contains an offset in an auxiliary table AlterTable which appears on the last page of 10PPST. Each entry in the latter is the address of a table of alteration sequences.

A.2.2.3 DoCrossJump

This routine performs the Cross Jump optimization. It scans through the code segment looking at only label nodes. When it finds one, it iterates through all nodes of the label's RefChain, doing two actions for each one. CrossJumpWork is called to compare the tail before the label against the tail before the JRST to it. Next CrossJumpWork is called to compare the tail of this JRST with the tails of every other JRST to the label. CrossJumpWork has an optional third argument which indicates which of these two cases is in effect, since it must do slightly different initialization. It is CrossJumpWork which is cognizant of the CJChain bit.

A.2.2.4 DoJump

This routine performs four optimizations. It makes a single scan over the code segment doing Unreachable, Jump Chain, Unused Label, and Label Merge. The code is straightforward and easy to follow.

A.2.2.5 DoLiterals

This routine makes a pass over the literals segment, which contains only string literals mentioned in the source program, deleting any such literal not referenced. There are two ways for a literal to appear here but not be referenced. The code referencing it may have been deleted by Unreachable, or the literal may appear in a manifest declaration of a name which is not used.

Finding such literals is quite easy. Labels are scanned for, and any label with an empty reference chain is deleted along with all further nodes up to the next label.

A.2.3 PeepFinal -- Bookkeeping Pass

Before the final code can be emitted into the .REL file, each instruction must be assigned its final PC value. An extra complication is that PC values must be known as the code is generated (long before optimization) so that entries can be made

in the symbol table. It is these data which permit BDDT to be able to print the source statement corresponding to any location in a running program. Since these PC values get changed as instructions are deleted, the symbol table must be adjusted. In passing over the code segment, PeepFinal generates a table (LocVec) of adjustments to be made. On completion of the pass, the routine SymtabAdjustILCA in MSYMB is called to make the actual changes. A comment at the beginning of SymtabAdjustILCA describes the format of the table. PeepFinal also assigns values to labels.